



AGH

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Fizyki i Informatyki Stosowanej

Praca magisterska

Łukasz Pawlik

kierunek studiów: **Informatyka Stosowana**

Opracowanie systemu szybkiego przesyłania danych z wykorzystaniem standardu USB

Promotor: **dr inż. Bartosz Mindur**

Kraków, 2015

Oświadczam, świadoma odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałam osobiście i samodzielnie i nie korzystałam ze źródeł innych niż wymienione w pracy.

.....

(czytelny podpis)

Merytoryczna ocena pracy przez Opiekuna:

Ocena końcowa pracy przez Opiekuna:

Data:

Podpis:

Merytoryczna ocena pracy przez Recenzenta:

Ocena końcowa pracy przez Recenzenta:

Data:

Podpis:

Spis treści

I. Wstęp	8
II. USB	9
II.1. Złącza	9
II.2. Historia	10
II.2.1. USB1.x	10
II.2.2. USB2.0	10
II.2.3. USB3.0	10
II.2.4. USB3.1	11
III. Biblioteki	12
III.1. libUSB	12
III.2. winUSB	12
III.3. porównanie libUSB oraz winUSB	13
IV. libUSB API	14
IV.1. najważniejsze struktury	14
IV.1.1. libusb_context	14
IV.1.2. libusb_device_handle	14
IV.2. najważniejsze funkcje	14
IV.2.1. libusb_init	14
IV.2.2. libusb_open_device_with_vid_pid	15
IV.2.3. libusb_kernel_driver_active	15
IV.2.4. libusb_detach_kernel_driver	15
IV.2.5. libusb_claim_interface	16
IV.2.6. libusb_bulk_transfer	16
IV.2.7. libusb_release_interface	17
IV.2.8. libusb_close	17
IV.2.9. libusb_exit	18
V. Mikrokontroler	19
VI. Synchroniczne przesyłanie danych	22
VI.1. zalety	22
VI.2. wady	22
VI.3. implementacja	22
VI.3.1. Funkcja getContext	22

VI.3.2. Funkcja <code>getDeviceHandler</code>	23
VI.3.3. Funkcja <code>proceedWithInitLibUsb</code>	23
VI.3.4. Funkcja <code>doTest</code>	24
VI.3.5. Funkcja <code>closeLibUsb</code>	25
VI.3.6. główny program synchronicznie przesyłający dane	26

I. Wstęp

Celem niniejszej pracy jest doświadczalne sprawdzenie czy możliwe jest uzyskanie prędkości większej bądź równej 140MBit/s czyli 17,5 MB/s.

W dokumencie zawarty jest prosty i zrozumiały opis, obrazujący różnice pomiędzy standardami USB. Znajduje się wprowadzenie do bibliotek umożliwiających korzystanie z API przygotowanego dla deweloperów chcących w łatwy i przystępny sposób korzystać z udogodnień portów USB.

W rozdziale "USB" znajduje się dokładny opis standardów USB jakie do dnia dzisiejszego zostały opracowane. Rozdział doskonale obrazuje różnice jakie z biegiem lat uwydatniły się, jak i chęć udoskonalenia standardu wpłynęła na jego dalszy rozwój.

W rozdziale "Biblioteki" przedstawiony został opis palety bibliotek które umożliwiają łatwy i szybki dostęp do portu USB. Najważniejsze z bibliotek zostały opisane w późniejszych rozdziałach.

W rozdziale "libUSB API" przedstawiony został dokładny opis wybranych funkcji biblioteki libUSB.

W rozdziale "Mikrokontroler" został przedstawiony dokładnie opis mikrokontrolera Land-Tiger LPC1768 na którym wykonane zostały testy i zostały zebrane dane potrzebne do tej pracy.

<TBD>

II. USB

Uniwersalna Magistrala Szeregowa jest to standard opracowany w latach 90. XX w. definiujący jakie kable, złącza oraz protokoły mają być używane podczas połączenia, komunikacji oraz definiuje sposób zasilania pomiędzy komputerem i urządzeniem elektronicznym.

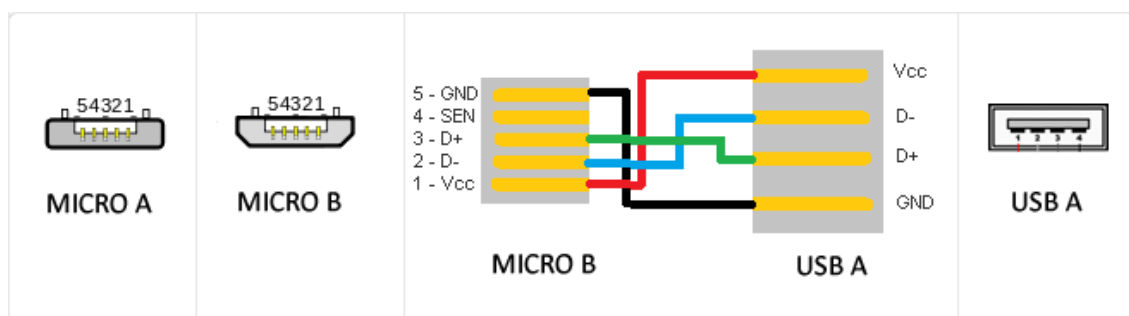
USB zostało zaprojektowane aby ułatwić połączenia standardowych elektronicznych urządzeń takich jak klawiatury, myszki, drukarki, aparaty cyfrowe, dyski przenośne do komputerów osobistych. Wszystkie te urządzenia są dodatkowo zasilane również za pomocą tego portu. Z czasem stało się to wspólne również dla innych urządzeń takich jak smartfony, palmtopy oraz konsole wideo.

USB szybko zastąpiło porty szeregowo oraz równoległe podobnie jak inne urządzenia zasilające elektroniczne urządzenia.

II.1. Złącza

Istnieją trzy podstawowe wielkości złączy USB. Najstarszy rozmiar (używany np. w pendrive'ach) występuje w standardach USB1.1, USB2.0, USB3.0, mini-USB (początkowo tylko dla złączy typu B, jak w wypadku wielu aparatów cyfrowych) oraz mikro-USB występuje również w trzech wariantach dla USB1.1, USB2.0, USB3.0 (dla przykładu używany w nowych telefonach komórkowych).

W przeciwieństwie do innych kabli do przesyłu danych (np. Ethernet, HDMI) każdy koniec kabla zakończony jest innym typem złącza (typem A lub typem B). Tylko złącze typu A dostarcza zasilanie. Zostało to zaprojektowane w taki sposób aby uniknąć elektrycznych przeciążeń a co za tym idzie uszkodzeniu urządzeniu. Istnieją również kable ze złączami typu A na obu końcach, ale nie należą do popularnych (i należy postępować z nimi ostrożnie). Kable USB mają zazwyczaj złącze typu A z jednej strony oraz złącze typu B z drugiej oraz wejście w komputerze lub urządzeniu elektronicznym. w przyjętej praktyce złącze typu A jest zazwyczaj największej (z możliwych wielkości), natomiast B w zależności od potrzeb użycia kabla (full, mini, micro).



Rys. II.1. Połączenie przewodów w micro-USB

II.2. Historia

USB zapoczątkowało w 1994 siedem firm: Compaq, DEC, IBM, Intel, Microsoft, NEC, Nortel. Celem było uproszczenie podłączenia zewnętrznych urządzeń do komputera zastępując stare złącza w płytach głównych wprowadzając rozwiązania na problemy znalezione w starych oraz upraszczając software. Pierwszy układ scalony wspierający USB został wyprodukowany przez Intel 1995r.

II.2.1. USB1.x

Pierwsza oficjalna wersja standardu USB została wydana w styczniu 1996r. USB1.0 charakteryzowała prędkość 1,5 Mbit/s (Low Speed) oraz 12 Mbit/s (Full Speed). Nie pozwalał jednak na używanie przedłużaczy kabli, wynikało to z limitów zasilania. Powstało kilka wypuszczono na rynek na chwile przed wydaniem standardu USB1.1 w sierpniu 1998r. W USB1.1 poprawiono kilka błędów znalezionych w USB1.0 i był to pierwszy standard, który został oficjalnie zaimplementowany w standardowych komputerach osobistych.

II.2.2. USB2.0

USB2.0 zostało wydane w kwietniu 2000r. udostępniając maksymalny przesył sygnału rzędu 480 Mbit/s (60MB/s) nazwany High Speed (USB1.x za pomocą Full Speed umożliwiał przesył rzędu 12Mbit/s). Biorąc pod uwagę zależności dostępu do magistrali przepustowość High Speed ogranicza się do 280 Mbit/s (35 MB/s).

Przyszłe modyfikacje do specyfikacji USB zostały zaimplementowane przez "Engineering Change Notices" (ECN). Najważniejsze z ECNów zostały dołączone do specyfikacji USB2.0 dostępnej na stronie internetowej USB.org.

Przykłady ECNow:

Złącze Mini-A oraz Mini-B: wydane w październiku 2000r.

II.2.3. USB3.0

Standard USB3.0 został wydany w listopadzie 2008r. definiujący zupełnie nowy tryb "SuperSpeed". Port USB zwyczajowo jest w kolorze niebieskim i kompatybilny z urządzeniami USB2.0 oraz kablami.

Dokładnie 17 listopada 2008r. ogłoszono iż specyfikacja dla wersji 3.0 została całkowicie ukończona i została zaakceptowana przez "USB Implementers Forum" (USB-IF), czyli głównej instytucji zajmującej się specyfikacjami standardu USB. To pozwoliło na szybkie udo-

stępnienie standardu deweloperom.

Nowa magistrała "SuperSpeed" dostarcza czwarty typ transferu z możliwością przesyłania sygnału z prędkością 5Gbit/s, ale poprzez użycie kodowania 8b/10b przepustowość wynosi 4Gbit/s. Specyfikacja uznaje za zasadne osiągnięcie prędkości w okolicach 3,2 Gbit/s (400 MB/s) co w założeniach powinno się zwiększać wraz z rozwijaniem hardware. Komunikacja odbywa się w obu kierunkach dla SuperSpeed (kierunek nie jest naprzemienny i nie jest kontrolowany przez hosta, jak to ma miejsce do wersji USB2.0).

Podobnie jak w poprzednich wersjach standardu, porty USB3.0 działają dwóch wariantach zasilania: niskiego poboru mocy (low-power: 150mA) oraz wysokiego poboru mocy (high-power: 900mA). Zapewniając odpowiedni jednocześnie pozwalają na przesył danych z prędkością SuperSpeed.

Została dodatkowo zdefiniowana specyfikacja zasilania (w wersji 1.2 wydana w w grudniu 2010r.) która zwiększała dopuszczalny pobór mocy do 1,5A, ale nie pozwala na współbieżne przesyłanie danych. Specyfikacja wymaga aby fizyczne porty same w sobie były w stanie obsłużyć 5A, ale ogranicza pobór do 1,5 A.

II.2.4. USB3.1

W styczniu 2013r. w prasie pojawiły się informacje o planach udoskonalenia standardu USB3.0 do 10Gbit/s. Zakończyło się to stworzeniem nowej wersji standardu - USB3.1. Wersja ta została wydana 31 lipca 2013r. wprowadzając szybszy typ przesyłania danych zwany "SuperSpeed USB 10 Gbit/s". Zaprezentowano również nowe logo stylizowane na zasadzie "Super-speed+". Standard USB3.1 zwiększył szybkość przesyłu sygnału do 10Gbit/s. Udało się też zredukować obciążenie łącza do 3% dzięki zmianie kodowania na 128b/132b.

Przy pierwszych testach prędkości USB3.1 udało się uzyskać prędkość 7,2Gbit/s.

Standard USB3.1 jest wstecznie kompatybilny ze standardem USB3.0 oraz USB2.0.

III. Biblioteki

III.1. libUSB

LibUSB jest biblioteką stworzoną w 2007 roku. Napisana w języku C pozwala na prosty i łatwy dostęp do urządzenia USB. Jest w 100% przeznaczona dla użytku developera. Biblioteka ma za zadanie ułatwić pisanie aplikacji opartych na komunikacji USB z mikrokontrolerem. Biblioteka libUSB jest przenośna a co za tym idzie dostępna na wiele platform (Linux, OS X, Windows, Android, OpenBSD, etc.) wraz z niezmiennym API. Nie są wymagane dodatkowe uprawnienia aby komunikacja z urządzeniem przebiegała poprawnie. Wspiera standardy USB:

- USB1.0
- USB1.1
- USB2.0
- USB3.0

Funkcjonalność biblioteki:

1. wszystkie typy transferu są wspierane (control, bulk, interrupt, isochronous)
2. 2 interfejsy
 - (a) synchroniczny (prosty)
 - (b) asynchroniczny (bardziej złożony ale bardziej efektywny)
3. stosowanie wątków jest bezpieczne
4. lekka biblioteka z prostym API
5. kompatybilna wstecznie (do wersji libUSB-0.1)

III.2. winUSB

Microsoft Windows począwszy od systemu Windows Vista wprowadził nowy zestaw bibliotek umożliwiający developerom korzystanie z portów USB. WinUSB udostępnia proste API, które pozwala aplikacji na bezpośredni dostęp do portów USB. Został stworzony w gruncie rzeczy dla prostych urządzeń obsługiwanych tylko przez jedną aplikację takich jak urządzenia do odczytu wskaźników pogodowych czy też innych programów które potrzebują szybkiego i bezpośredniego dostępu do portu. WinUSB udostępnia API aby odblokować developera przy pracy z portami USB z poziomu user-mode. W Windowsie 7 USB Media Transfer

Protocol (MTP) używa winUSB zamiast poprzednio stosowanych rozwiązań kernela (kernel mode filter driver).

Media Transfer Protocol jest rozszerzeniem PTP (Picture Transfer Protocol) i jest protokołem pozwalającym na przesyłanie atomowe plików audio oraz wideo z oraz do urządzenia. PTP początkowo został zaprojektowany do ściągania zdjęć, obrazów z aparatów cyfrowych, Media Transfer Protocol pozwala na przesyłanie plików muzycznych z cyfrowych urządzeń odtwarzających muzykę oraz pliki video z urządzeń pozwalających na ich odtworzenie.

MTP jest częścią frameworku "Windows Media" blisko związanym z odtwarzaczem Windows Media Player. Systemy Windows począwszy od Windows XP SP2 wspierają MTP. Windows XP wymaga Windows Media Player w wersji 10 lub wyższej, późniejsze wersje systemu wspierają już go domyślnie. Microsoft posiada dodatkowo możliwość zainstalowania MTP na wcześniejszych wersjach systemu ręcznie do wersji Microsoft Windows 98.

Twórcy standardu USB ustandaryzowali MTP jako pełnoprawną klasę dla urządzeń USB w maju 2008r. Od tamtej pory MTP jest oficjalnym rozszerzeniem PTP i współdzieli ten sam kod klasy.

III.3. porównanie libUSB oraz winUSB

IV. libUSB API

IV.1. najważniejsze struktury

IV.1.1. libusb_context

libusb_context jest strukturą reprezentującą sesję libusb.

Koncepcja indywidualnych sesji libusb pozwala aby program mógł korzystać z dwóch bibliotek (lub dynamicznie ładować dwa moduły) z których obie nie zależnie korzystają z libusb. To zapobiega ingerencji (interferencji) pomiędzy dwoma programami używającymi libusb. Dla przykładu libusb_set_debug() nie zaingeruje w działanie innego programu korzystającego z libusb, natomiast libusb_exit() nie wyczyści pamięci używanej przez inny program libusb.

Sesje tworzone są za pomocą libusb_init() oraz czyszczone za pomocą libusb_exit(). Jeśli zagwarantowane jest to, że dana aplikacja jest jedyną która korzysta z libusb, twórca jej nie musi przejmować się kontekstami (strukturą libusb_context), wystarczy aby przekazywał do wszystkich funkcji, gdzie struktura jest wymagana wartość NULL. Jest to równoważne z użyciem domyślnego kontekstu.

IV.1.2. libusb_device_handle

Struktura reprezentująca uchwyt do urządzenia USB.

Jest to nieprzejrzysty typ, użycie jest możliwe tylko za pomocą wskaźnika, zazwyczaj dostarczanego za pomocą funkcji libusb_open().

Uchwyt do urządzenia USB jest używany do wykonywania operacji wejścia/wyjścia. Po zakończeniu wszystkich operacji należy wywołać libusb_close().

IV.2. najważniejsze funkcje

IV.2.1. libusb_init

Inicjalizacja biblioteki.

Funkcja musi zostać wywołana przed wywołaniem jakiejkolwiek innej funkcji z biblioteki libusb.

Jeśli w argumencie nie zostanie dostarczony żaden wskaźnik (wyjściowy) kontekstu, zostanie stworzony domyślny kontekst. W przypadku jeśli domyślny kontekst już istnieje zostanie on ponownie użyty (bez ponownej inicjalizacji).

Funkcja zwraca wartość 0 w wypadku powodzenia, w przeciwnym wypadku zwraca kod błędu.

IV.2.2. `libusb_open_device_with_vid_pid`

Wygodna funkcja służąca odszukaniu konkretnego urządzenia na podstawie jego `vendorId` oraz `productId` (są to parametry charakteryzujące każde urządzenie).

Ta funkcja jest używana w przypadkach kiedy z góry jest znane `vendorId` oraz `productId`. Najczęściej są to przypadki pisania aplikacji aby przetestować jakąś określoną funkcjonalność. Funkcja pozwala uniknąć wywołanie `libusb_get_device_list()` i dbania o odpowiednie czyszczenie pamięci po liście.

Pierwszym parametrem jest kontekst uzyskany za pomocą `libusb_init()`.

Kolejnymi parametrami są `vendorId` oraz `productId`.

Funkcja zwraca uchwyt do znalezionej urządzenia, lub `NULL` w wypadku kiedy nie może znaleźć pożądanego urządzenia (o podanym `productId` oraz `vendorId`) lub błędu.

IV.2.3. `libusb_kernel_driver_active`

Funkcja sprawdzająca czy sterownik jądra `kernela` jest aktywny na interfejsie.

W wypadku kiedy sterownik jądra jest aktywny nie możliwe jest zgłoszenie użycia interfejsu, a co za tym idzie `libUSB` nie może wykonać operacji wejścia/wyjścia.

Funkcjonalność jest nie dostępna w systemie `Windows`.

Do funkcji należy przekazać uchwyt do urządzenia oraz numer interfejsu.

Funkcja zwraca wartość 0, jeśli żaden sterownik jądra nie jest aktywny, wartość 1 w wypadku jeśli istnieje aktywny sterownik jądra.

W wypadku błędów: `LIBUSB_ERROR_NO_DEVICE` jeśli urządzenie zostanie odłączone, `LIBUSB_ERROR_NOT_SUPPORTED` dla platform, gdzie funkcjonalność nie jest wspierana oraz inny kod błędu w wypadku innego błędu.

IV.2.4. `libusb_detach_kernel_driver`

Dzięki funkcji możliwe jest odłączenie sterownika jądra `kernela` od interfejsu.

Sukces operacji umożliwi zgłoszenie użycia interfejsu i wykonanie operacji wejścia/wyjścia.

Funkcjonalność nie jest dostępna dla systemu `Windows`.

Parametrami są uchwyt do urządzenia oraz numer interfejsu.

Funkcja zwraca wartość 0 w momencie powodzenia, `LIBUSB_ERROR_NOT_FOUND` jeśli żaden sterownik nie był aktywny, `LIBUSB_ERROR_INVALID_PARAM` jeśli interfejs nie istnieje,

`LIBUSB_ERROR_NO_DEVICE` jeśli urządzenie zostało odłączone, `LIBUSB_ERROR_NOT_SUPPORTED`

dla platform, gdzie funkcjonalność nie jest wspierana lub inny kod błędu w wypadku innego błędu.

IV.2.5. **libusb_claim_interface**

Dzięki funkcji `libusb_claim_interface()` możliwe jest zgłoszenie użycia danego interfejsu dla danego urządzenia.

Wywołanie funkcji jest wymagane przed wykonaniem operacji wejścia/wyjścia dla dowolnego punktu końcowego interfejsu.

Jest dozwolone wywołanie funkcji dla interfejsu już wcześniej zgłoszonego, w tym wypadku zostanie zwrócona wartość 0 bez wykonywania żadnych operacji.

W wypadku jeśli zmienna `auto_detach_kernel_driver` jest ustawiona na wartość 1 dla danego urządzenia (zmienna ustawiana za pomocą funkcji: `libusb_set_auto_detach_kernel_driver()`) sterownik jądra zostanie odłączony (jeśli to konieczne), w przypadku niepowodzenia zostanie zwrócony błąd odłączenia.

Sama procedura wewnątrz funkcji nie jest skomplikowana, nie wymaga wysyłania czegokolwiek po magistrali. Są to proste instrukcje mówiące systemowi operacyjnemu iż aplikacja chce korzystać z danego interfejsu.

Nie jest to funkcja blokująca.

Parametrami są uchwyt do urządzenia oraz numer interfejsu zgłaszanego.

Wartość 0 zostaje zwrócona w wypadku powodzenia operacji.

W wypadku niepowodzenia kody błędu t.j. `LIBUSB_ERROR_NOT_FOUND` jeśli podany interfejs nie istnieje, `LIBUSB_ERROR_BUSY` jeśli inny program lub sterownik zarezerwował dany interfejs, `LIBUSB_ERROR_NO_DEVICE` jeśli urządzenie zostało odłączone oraz inne.

IV.2.6. **libusb_bulk_transfer**

Dzięki funkcji możliwy jest przesył większej grupy danych.

Kierunek jest określony na podstawie bitów kierunkowych punktu końcowego interfejsu.

Dla odczytu, jeden z parametrów określa ilość danych jaka jest spodziewana przy odczycie. Jeżeli odebrana zostanie mniejsza ilość danych niż oczekiwana, funkcja po prostu zwróci te dane wraz z dodatkowym parametrem określającym ilość otrzymanych danych. Istotne przy odczycie jest to aby sprawdzić czy ilość oczekiwanych danych jest taka jak odczytana.

W wypadku zapisu również należy sprawdzić czy ilość danych wysłanych pokrywa się z ilością danych skierowaną do wysyłki.

Wskazana jest również weryfikacja ilości danych wysłanych/odebranych w wypadku wystąpienia timeoutu (funkcja zwróci kod błędu określający timeout). `libUSB` może podzielić wysyłane dane na mniejsze części i timeout może wystąpić po wysłaniu kilku z nich. Ważne jest

to, że nie oznacza to iż nic nie zostało wysłane/odebrane, dlatego należy sprawdzić ilość elementów wysłanych/odebranych i dostosować odpowiednio kolejne kroki.

Funkcja przyjmuje następujące parametry: uchwyt urządzenia z którym aplikacja będzie się komunikować, adres punktu końcowego interfejsu po którym będzie odbywała się komunikacja, wskaźnik do pamięci danych która ma zostać przetransferowana (w wypadku zapisu) lub odebrana (w wypadku odczytu), ilość danych do wysłania (w przypadku zapisu) lub oczekiwana ilość danych do odebrania (w przypadku odczytu), ilość danych przetransferowanych (w obu przypadkach), maksymalna długość czasu na wykonanie operacji, dla nieograniczonego należy użyć wartości równej 0.

W przypadku poprawności działania funkcja zwraca wartość 0 oraz ilość przetransferowanych danych przekazanych do funkcji za pomocą wskaźnika.

W przeciwnym wypadku funkcja zwraca: `LIBUSB_ERROR_TIMEOUT` jeśli transfer przekroczył określony czas, `LIBUSB_ERROR_PIPE` jeśli wystąpił błąd związany z punktem końcowym, `LIBUSB_ERROR_OVERFLOW` jeśli urządzenie wysłało więcej danych niż przewidziane w buforze, `LIBUSB_ERROR_NO_DEVICE` jeśli urządzenie zostało odłączone oraz inne.

IV.2.7. `libusb_release_interface`

Funkcja zwalnia rezerwację wcześniej zgłoszonego interfejsu za pomocą funkcji `libusb_claim_interface()`. Zwolnienie wszystkich interfejsów jest wymagane przed zamknięciem urządzenia.

Nie jest to blokująca funkcja.

W wypadku jeśli zmienna `auto_detach_kernel_driver` jest ustawiona na wartość 1 dla danego urządzenia (zmienna ustawiana za pomocą funkcji: `libusb_set_auto_detach_kernel_driver()`) sterownik jądra zostanie ponownie podłączony zaraz po zwolnieniu interfejsu.

Parametrami są: uchwyt urządzenia oraz numer interfejsu dla niego poprzednio zarezerwowanego.

Metoda zwraca 0 gdy wszystkie operacje się powiodą.

W przeciwnym wypadku zwraca: `LIBUSB_ERROR_NOT_FOUND` jeśli interfejs nie został poprzednio zarezerwowany (zgłoszony do użycia), `LIBUSB_ERROR_NO_DEVICE` jeśli urządzenie zostało odłączone oraz inne.

IV.2.8. `libusb_close`

Zwalnia uchwyt do urządzenia.

Funkcja powinna być wołana na wszystkich używanych uprzednio uchwytach.

Funkcja pokrótce niszczy referencje stworzoną za pomocą `libusb_open()` dla danego urządzenia.

Jest to funkcja nie blokująca.

Parametrem jest uchwyt przeznaczony do zamknięcia.

IV.2.9. libusb_exit

Funkcja zamyka dostęp do biblioteki.

Powinna być wołana po zamknięciu wszystkich otwartych urządzeń ale przed zakończeniem działania programu.

Parametrem jest kontekst który ma zostać zamknięty, w wypadku wartości NULL wybierany jest domyślny.

V. Mikrokontroler



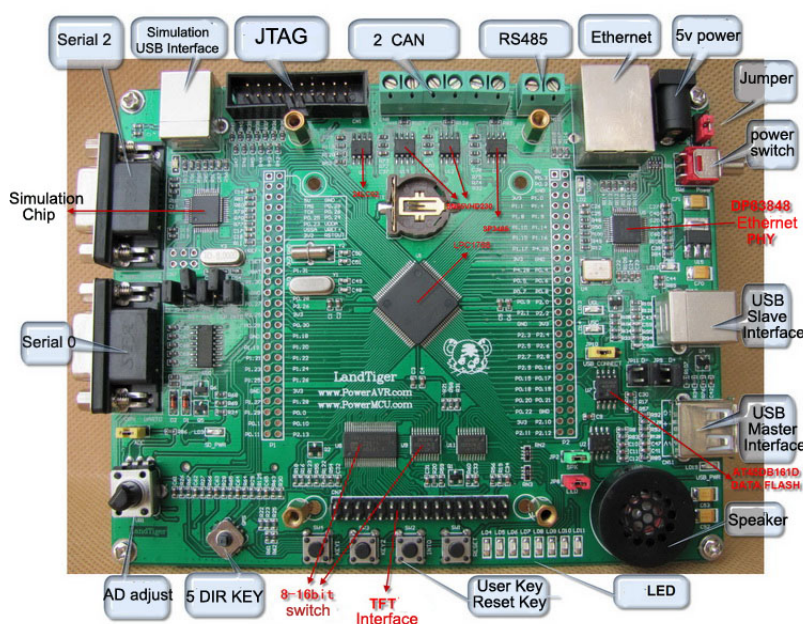
Rys. V.1. Mikrokontroler LandTiger wraz z wyświetlaczem

Mikrokontroler LandTiger oparty na LPC1768 został wyprodukowany przez firmę PowerMCU i można ją zakupić od wielu dostawców na eBay lub innych serwisach świadczących usługi zakupów przez internet. Średni koszt waha się w granicach \$70 za płytkę wraz z wyświetlaczem LCD 3,2 cala o rozdzielczości 320x240 pikseli, z zasilaczem oraz zestawem kabli.

Funkcjonalności:

- (a) 2 porty RS232, jeden z nich wspiera ISP (In-system Programming)
- (b) 2 interfejsy magistrali CAN (Controller Area Network)
- (c) interfejs RS485
- (d) interfejs Ethernetowy RJ45-10/100M
- (e) przetwornik cyfrowo-analogowy (DAC) wraz z wmontowanym głośnikiem (wyjściem interfejsu) oraz sterownikiem dźwięku (LM386)
- (f) przetwornik analogowo-cyfrowy (ADC) wraz z wbudowanym potencjometrem (wejściem interfejsu).

- (g) Kolorowy 3,2 cala (lub 2,8 cala) dotykowy wyświetlacz LCD o rozdzielczości 320x240 pikseli.
- (h) interfejs USB2.0 (USB Host oraz USB Device)
- (i) interfejs kard SD/MMC
- (j) interfejs I2C połączony z 2Kbit pamięcią EEPROM
- (k) interfejs SPI połączony z 16Mbit pamięcią flash
- (l) 2 user keys, 2 function keys
- (m) 8 diód typu LED
- (n) pięciokierunkowy joystick
- (o) wsparcie dla pobierania ISP
- (p) pobieranie z użyciem JTAG, interfejs dla debugowania
- (q) zintegrowany emulator kompilacji JLINK - wspiera możliwość debugowania online (po kablu USB podłączonym do PC) dla środowisk deweloperskich tj. KEIL, IAR, CooCox i innych
- (r) dodatkowe 5V port zasilający (możliwe jest też za pomocą portu USB)



Rys. V.2. Mikrokontroler LandTiger wraz z opisem poszczególnych elementów

LandTiger jest oparty na LPC1768. Wbudowany hardware wspiera ISP aby umożliwić załadowanie kodu (z użyciem bin2hex oraz flashmagic).

Alternatywą jest to, że kod może zostać załadowany za pomocą emulatora JLINK JTAG/SWD lub za pomocą zewnętrznego urządzenia JTAG.

Port COM1 (UART0) wspiera komunikację z PC w obie strony. Wszelkie funkcje portu USB są wspierane z minimalnymi zmianami w oprogramowaniu. Podobnie jest z Ethernetem, z niewielkimi zmianami w oficjalnym kodzie dla LPC1768 kod jest w stanie się uruchomić na LandTigerze.

Wyświetlacz LCD jest oparty na kontrolerze SSD1289. Wyświetlacz może zostać odłączony od płyty. Używa 8-bitowej magistrali $P_{20}..P_{27}$. Kontroler ekranu dotykowego jest dostarczony razem z modułem wyświetlacza. Interfejs pomiędzy ekranem dotykowym a LPC1768 jest możliwy dzięki SPI.

Główne różnice pomiędzy LandTigerem a LPC1768:

- (a) płyta po podłączeniu do PC nie pokazuje się jako zewnętrzne urządzenie magazynujące
- (b) aby ściągnąć nowe pliki binarne należy użyć ISP lub JTAG
- (c) brak wsparcia dla serialowego portu po linku USB, należy używać RS232 lub portu USB
- (d) brak wsparcia dla logicznego systemu plików
- (e) brak wsparcia dla 4 diód typu LED (istnieje możliwość użycia innych).

VI. Synchroniczne przesyłanie danych

Jedną z metod dostępnych, która została użyta w przygotowaniu tej pracy jest wykorzystanie synchronicznych możliwości przesyłu danych z wykorzystaniem libUSB.

VI.1. zalety

- duża kontrola nad wysyłanymi/odbieranymi danymi
- debugowanie jest mało skomplikowane w porównaniu do asynchronicznego przesyłania danych

VI.2. wady

- możliwości prędkości są mocno ograniczone poprzez funkcje blokujące

VI.3. implementacja

Implementacja opiera się na wykorzystaniu prostych podstawowych funkcji z biblioteki libUsb.

VI.3.1. Funkcja getContext

W listingu VI.1 przedstawione zostało ciało funkcji getContext().

Funkcja nie przyjmuje żadnego argumentu, natomiast zwraca kontekst który jest później przekazywany do wielu metod jako argument. W rozdziale IV wspomniane zostało iż możliwe jest używanie kontekstu domyślnego, wtedy zamiast tej funkcji i wartości zwracanej (pозyskanego kontekstu) w każdym użyciu uzyskanego przez getContext() wskaźnika należałoby wpisać wartość NULL.

Aby używać domyślnego kontekstu należy mieć pewność iż aplikacja/wątek jest jedynym użytkownikiem libUSB.

```
libusb_context* getContext()
{
    libusb_context* ctx = NULL;
    int r = libusb_init(&ctx);
    if (r < 0) {
        std::cout<<"Init Context error Error "<< r <<std::endl;
        return NULL;
    }
}
```

```
}  
    return ctx;  
}
```

Listing VI.1. Funkcja getContext()

VI.3.2. Funkcja getDeviceHandler

Funkcja nie przyjmuje żadnego argumentu. Zwracany jest uchwyt do LandTiger'a, lub wartość NULL w wypadku błędu (np. braku podłączonego i uruchomionego mikrokontrolera). Ciało funkcji zostało przedstawione w Listingu VI.2

```
libusb_device_handle* getDeviceHandle(libusb_context* ctx)  
{  
    libusb_device_handle* dev_handle = libusb_open_device_with_vid_pid(ctx,  
        LAND_TIGER_VID, LAND_TIGER_PID);  
    if(dev_handle == NULL)  
        std::cout<<"Cannot open device"<<std::endl;  
    else  
        std::cout<<"Device Opened"<<std::endl;  
  
    return dev_handle;  
}
```

Listing VI.2. Funkcja getDeviceHandler()

VI.3.3. Funkcja proceedWithInitLibUsb

Funkcja odpowiedzialna za dokończenie procedur inicjalizacyjnych. Ciało funkcji zostało przedstawione w Listingu VI.3. Funkcja sprawdza dodatkowo czy sterownik jądra kernela jest aktywny, dla platformy Windows funkcja zwróci wartość LIBUSB_ERROR_NOT_SUPPORTED (!= 1) i zachowa się analogicznie jak w wypadku nie aktywnego sterownika w systemie UNIX (warunek będzie nie spełniony).

Kolejnym krokiem jest rezerwacja przez program konkretnego interfejsu za pomocą funkcji libusb_claim_interface.

```
int proceedWithInitLibUsb(libusb_device_handle* dev_handle, libusb_context* ctx)  
{  
  
    if(libusb_kernel_driver_active(dev_handle, 0) == 1) { //find out if kernel driver is  
        attached  
        std::cout<<"Kernel Driver Active"<<std::endl;  
        if(libusb_detach_kernel_driver(dev_handle, 0) == 0) //detach it  
            std::cout<<"Kernel Driver Detached!"<<std::endl;  
    }
```

```
}  
int status = libusb_claim_interface(dev_handle, 1);  
if(status < 0)  
{  
    std::cout<<"Cannot Claim Interface"<<std::endl;  
    return 1;  
}  
std::cout<<"Claimed Interface"<<std::endl;  
  
return 0;  
}
```

Listing VI.3. Funkcja `proceedWithInitLibUsb()`

VI.3.4. Funkcja `doTest`

Funkcja w całości przedstawiona w Listingu VI.4 i jest odpowiedzialna za wykonanie podstawowego pomiaru czasu przepływu danych w obie strony pomiędzy PC a mikrokontrolerem. Została zaprojektowana w taki sposób aby konkretny bufor danych został wysłany oraz odebrany określoną ilość razy i zwrócony przedział czasowy w jakim udało się to uzyskać. Powodem wysyłania/odbierania danych określoną ilość razy jest fakt dość sporych ograniczeń jeśli chodzi o chip wbudowany w płytę LandTiger.

```
int doTest(libusb_device_handle* dev_handle, int bufferSize, int count, double*  
    timeResult)  
{  
    unsigned char *data_out = new unsigned char[bufferSize]; //data to write  
    unsigned char* data_in = new unsigned char[bufferSize];  
    generateSimulatedData(data_out, bufferSize);  
    int howManyBytesIsSend;  
    int howManyBytesReceived;  
  
    time_t start_t, end_t;  
    *timeResult = 0;  
  
    time(&start_t);  
    for(int i = 0; i < count; ++i)  
    {  
        int sendStatus = libusb_bulk_transfer(dev_handle, (2 | LIBUSB_ENDPOINT_OUT),  
            data_out, bufferSize, &howManyBytesIsSend, 0);  
        if(sendStatus == 0 && howManyBytesIsSend == bufferSize)  
        {  
            //here was printing data for debugging only
```



```
}  
else  
{  
    std::cout<< "Write Error" << std::endl;  
    delete [] data_out;  
    return 1;  
}  
  
int readStatus = libusb_bulk_transfer(dev_handle, (2 | LIBUSB_ENDPOINT_IN),  
data_in, bufferSize * sizeof(unsigned char), &howManyBytesReceived, 0);  
if (readStatus == 0 && howManyBytesReceived == howManyBytesIsSend)  
{  
    //here was printing data for debugging only  
}  
else  
{  
    std::cout << "Read Error: " << readStatus << std::endl;  
    delete [] data_in;  
    return 1;  
}  
  
}  
time(&end_t);  
*timeResult = difftime(end_t, start_t);  
delete [] data_in;  
delete [] data_out;  
return 0;  
}
```

Listing VI.4. Funkcja doTest()

VI.3.5. Funkcja closeLibUsb

Funkcja przedstawiona w Listingu VI.5 odpowiada za zwolnienie interfejsu oraz zasobów uprzednio zajętych na czas testu.

```
int closeLibUsb(libusb_device_handle* dev_handle, libusb_context* ctx)  
{  
    int status = libusb_release_interface(dev_handle, 1);  
    if(status != 0) {  
        std::cout<<"Cannot Release Interface"<<std::endl;  
        return 1;  
    }  
    std::cout<<"Released Interface"<<std::endl;  
}
```

```
libusb_close(dev_handle);  
libusb_exit(ctx);  
return 0;  
}
```

Listing VI.5. Funkcja closeLibUsb()

VI.3.6. główny program synchronicznie przesyłający dane

Kod przedstawiony w Listingu VI.6 doskonale ukazuje prostotę korzystania z libUSB. Użytkownik zobligowany jest do wprowadzenia wielkości bufora danych oraz ilości powtórzeń określających ile razy dany bufor zostanie wysłany do mikrokontrolera. Następnie zostaje wykonana inicjalizacja z użyciem wyżej wymienionych funkcji oraz test główny również opisany powyżej. Całość zostaje zakończona czyszczeniem zarezerwowanych zasobów. W kodzie widoczne są wszelkiego rodzaju komunikaty o błędach oraz przy inicjalizacji aby użytkownik miał świadomość jak wielkich rozmiarów test będzie wykonywany.

```
int main(int argc, char* argv[])  
{  
    if(argc < 3)  
    {  
        std::cout << "use: libusbttest <bufforSize> <count>" << std::endl;  
        std::cout << "Note that max buffer of LandTiger is " << BUFFER_MAX << "Bytes" <<  
        std::endl;  
        return 0;  
    }  
  
    int bufforSize = atoi(argv[1]);  
    if(bufforSize > BUFFER_MAX)  
    {  
        std::cout << "bufforSize is grather than 64B, setting 64 as default" << std::endl;  
        bufforSize = BUFFER_MAX;  
    }  
  
    int count = atoi(argv[2]);  
  
    std::cout << "Total size to send/receive: " << bufforSize << " x " << count << " = "  
        << bufforSize * count << " Bytes" << std::endl;  
  
    libusb_context *ctx = getContext();  
    libusb_device_handle* dev_handle = getDeviceHandle(ctx);  
    if(ctx == NULL || dev_handle == NULL)  
    {
```

```
        return 1;
    }
    int initStatus = proceedWithInitLibUsb(dev_handle, ctx);
    if(initStatus != 0)
    {
        std::cout << "proceedWithInitLibUsb exited with errors!" << std::endl;
        return 1;
    }
    double testResult = 0.;
    int testStatus = doTest(dev_handle, bufforSize, count, &testResult);
    if(testStatus != 0)
    {
        std::cout << "There was an error during tests!!" << std::endl;
    }
    else
    {
        std::cout << "Sending of: " << bufforSize * count << "Bytes using bufferSize=" <<
        bufforSize << " takes " << testResult << "s." << std::endl;
    }

    if(closeLibUsb(dev_handle, ctx) != 0)
    {
        return 1;
    }

    return 0;
}
```

Listing VI.6. Funkcja *main()*